GD2S03
Advanced Software Engineering & Programming for Games

Bachelor of Software Engineering(BSE) Game Development

- Overview
  - Initialization
  - Deinitialization
  - Optional Chaining
  - Error handling
  - Type Casting

Initialization

- *Initialization* is the process of preparing an instance of a class, structure, or enumeration for use

- Classes and structures *must* set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created.

- Stored properties cannot be left in an indeterminate state.

- **Property observers** are not called
    - when default value is assigned to a stored property, or
    - set its initial value within an initializer as the value of that property is set directly.

Initialization

Initializers

- *Initializers* are called to create a new instance of a particular type written using the init keyword

```
init() {
        // perform some initialization here
    }
```

- Default Property Values
- *Initial value of the stored property can be set inside initializer*

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

Default property value can be set as part of the property's declaration

```
struct Fahrenheit {
    var temperature = 32.0
}
```

Initialization – Customizing Initializing

Initialization Parameters

- *initialization parameters* can be provided as part of an initializer's definition.
- Use (_) underscore, if parameter label is not required
- Stored property can be declared optional if its value cannot be set during initialisation.
- For class instances, a constant property can be modified during initialization only by the class that introduces it. It cannot be modified by a subclass.

```
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }

    func ask() {
        print(text)
    }
}


let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask() // Prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with
cheese.)"
```

Initialization – Default Initializers

- Swift provides a *default initializer* for any structure or class that provides default values for all of its properties and
- Swift does not provide any default initializer if default values are not provided to all of its properties.
- Unlike a default initializer, the structure receives a **member wise** initializer even if it has stored properties that do not have default values.
- The memberwise initializer is a shorthand way to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name.

```
struct Size{
    var width:Int      //need to have
    var height:Int     //default  value
                       //for class if default
}                      //initializer
                       //is required

let size = Size(width:10, height:10)
print("area is \(size.width * size.height)")

----------------------------------------
struct Size{
    var width:Int = 10
    var height:Int = 20
}

let defaultSize = Size()
print("area is \(defaultSize.width *
defaultSize.height)")
```

Initialization – Initialization Delegation for value Types

- Initializers can call other initializers to perform part of an instance's initialization known as *initializer delegation*.
- Value types (structures and enumerations) can only delegate to another initializer that they provide themselves because they don't allow inheritance.
- For value types, use ***self.init*** to refer to other initializers.
- Default initializer or member wise initializer is not provided if a custom initializer is present, otherwise use ***Extension***

```
struct Size{
    var width:Int = 0
    var height:Int = 0

    init() { }
    init(_ width:Int, _ height:Int){
        self.width = width
        self.height = height
    }
    init(w width:Int, h height:Int){
        self.init(width, width)
    }
}
var size1 = Size()
var size2 = Size(13, 13)
var size3 = Size(w:15, h:15)

print("Dafault area is \(size1.width * size1.height)")
print("Area from no label initializer is \(size2.width *
size2.height)")
print("Area after labelled initializer is \(size3.width *
size3.height)")
```

Initialization – Class Inheritance and Initialization

- All of a class's stored properties—including any properties the class inherits from its superclass—*must* be assigned an initial value during initialization.
- Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as designated initializers and convenience initializers.

- *Designated initializers* are the primary initializers for a class.
- A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.
- Classes tend to have very few designated initializers, and it is quite common for a class to have only one.
- Every class must have at least one designated initializer.

- *Convenience initializers* are secondary, supporting initializers for a class.
- Convenience initializers are used to call a designated initializer from the same class as the with some of the designated initializer's parameters set to default values.
- A Convenience initializer is optional.
- Starts with keyword "convenience"

Initialization - Rules

To simplify the relationships between designated and convenience initializers, Swift applies the following three rules for delegation calls between initializers:

Rule 1

- **A designated initializer must call a designated initializer from its immediate superclass.**

Rule 2

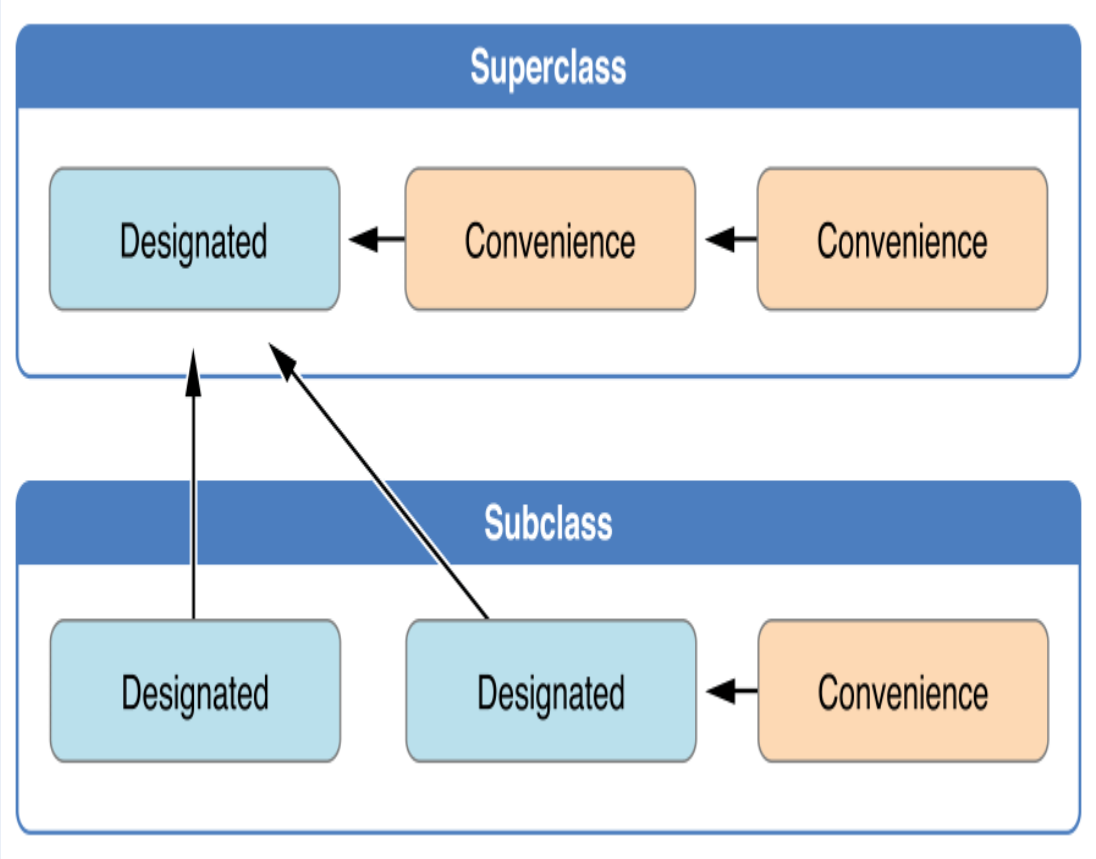
- **A convenience initializer must call another initializer from the *same* class.**

Rule 3

- **A convenience initializer must ultimately call a designated initializer.**

A simple way to remember this is:

- Designated initializers must always delegate *up*.
- Convenience initializers must always delegate *across*.



Initialization – Safety Check

- Swift's compiler performs four helpful safety-checks to make sure that two-phase initialization is completed without error:

Safety check 1

- A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.

Safety check 2

- A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

Safety check 3

- A convenience initializer must delegate to another initializer before assigning a value to any property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

Safety check 4

- An initializer cannot call any instance methods, read the values of any instance properties, or refer to self as a value until after the first phase of initialization is complete.

Initialization – 2 phase of Initialization

Phase 1

- A designated or convenience initializer is called on a class.
- Memory for a new instance of that class is allocated. The memory is not yet initialized.
- A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized.
- The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties.
- This continues up the class inheritance chain until the top of the chain is reached.
- Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

Phase 2

- Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further.
- Initializers are now able to access self and can modify its properties, call its instance methods, and so on.
- Finally, any convenience initializers in the chain have the option to customize the instance and to work with self.

Initialization – Inheritance and Overriding

- Use keyword “override” to override the superclass initializer.
- Override keyword is not used when implementing the matching convenience initializer from superclass. (why?)
- In this example, Bicycle class overrides the default initializer from the super class.

```
class Plant {
    var color = "BLACK"
    var description: String {
        return "\(color) Color"
    }
}
class Rose: Plant {
    var petals: Int
    override init() {
        self.petals = 5             //Rule 1
        super.init()                //Rule 2
        color = "WHITE"
    }
    convenience init(_color: String){
        self.init()                 //Rule 3
        color = _color
    }
}
let rose = Rose()
print(rose.description)

let rareRose = Rose(_color: "diamond blue")
print(rareRose.description)
```

Initialization – Automatic Initializer Inheritance

- Subclass does not inherit their superclass initializers by default, in certain condition it can -

Rule 1

- If a subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

Rule 2

- If a subclass provides an implementation of all of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

- **RecipeIngredient** has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore **RecipeIngredient** automatically inherits all of its superclass's convenience initializers too.

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}


class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

Initialization

- All three of these initializers can be used to create new **RecipeIngredient** instances:

- **ShoppingListItem** automatically inherits *all* of the designated and convenience initializers from its superclass as it provides a default value for all of the properties it introduces and does not define any initializers itself.
- All three of the inherited initializers can be used to create a new `ShoppingListItem` instance

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)


class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

Initializers - Failable

- Sometimes it's required initializers to fail, because of invalid initialization or some other condition.
- Failable initializer is written as "init?"
- A failable and nonfailable initializer with the same parameter cannot be defined.
- Failable initializer may return nil to indicate failure.

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}


let someCreature = Animal(species: "Giraffe")
let anonymousCreature = Animal(species: "")
if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}
```

Initialization – Overriding a failable Initializer

- A subclass can override the failable initializer of superclass as a failable or non failable initializer
- If a subclass overrides failable superclass initializer as nonfailable, then the only way to delegate up to the superclass initializer is to force unwrap the result of the failable superclass initializer
- A subclass can implement failable initializer as nonfailable initializer but not other way around

```
class Document {
    var name: String?
/*this initializer
creates a document with
a nil name value*/
    init() {}
   /*this initializer
creates a document with
a nonempty name value */
    init?(name: String){
       if name.isEmpty
       {
          return nil
        }
        self.name = name
    }
}
```

```
class AutomaticallyNamedDocument:
Document {
    override init() {
      super.init()
      self.name = "[Untitled]"
    }
    override init(name: String){
    super.init()
    if name.isEmpty {
        self.name = "[Untitled]"
    } else {
        self.name = name
      }
    }
}
```

```
class UntitledDocument: Document {
   override init(){
     super.init(name:"[Untitled]")!
   }
}
```

Initialization – init! failable Initializer

- You typically define a failable initializer that creates an optional instance of the appropriate type by placing a question mark after the init keyword (init?).
- Alternatively, you can define a failable initializer that creates an implicitly unwrapped optional instance of the appropriate type.
-
- Do this by placing an exclamation mark after the init keyword (init!) instead of a question mark.
- You can delegate from init? to init! and vice versa, and you can override init? with init! and vice versa.
- You can also delegate from init to init!, although doing so will trigger an assertion if the init! initializer causes initialization to fail

Initialization - Required

- `required` modifier used before initializer makes it mandatory for every subclass to implement it.

```
class SomeClass {
    required init() {
    // initializer implementation goes here
    }
}
```

- `required` modifier must be written before every subclass implementation of a required initializer, to indicate that the initializer requirement applies to further subclasses in the chain.
- Do not write the override modifier when overriding a required designated initializer:

```
class SomeSubclass: SomeClass {
    required init() {
    // subclass implementation of the
required initializer goes here
    }
}
```

- A closure or a global function can be used to provide some customized default value to a property.
- These type of closures and function create temporary value of the same type and then return it.
- Parentheses tells swift to execute the closure

```
class SomeClass {
    let someProperty: SomeType = {
    /* create and return default value
       for SomeType */
    return someValue
    }()
}
```

Deinitialization

- Class definitions can have at most one deinitializer per class.
- The deinitializer does not take any parameters and is written without parentheses.
- Deinitializers are called automatically, just before instance deallocation takes place.
- You are not allowed to call a deinitializer yourself. Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer implementation.
- Superclass deinitializers are always called, even if a subclass does not provide its own deinitializer.
- Because an instance is not deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it is called on and can modify its behavior based on those properties (such as looking up the name of a file that needs to be closed)

```
deinit {
    // perform the deinitialization
}
```

Optional Chaining

- *A* process for querying and calling properties, methods, and subscripts on an optional that might currently be **nil.**
- Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is **nil**.
- A property that normally returns an **Int** will return an **Int?**
- Force unwrapping a nil optional value will trigger a runtime error.
- Question mark must be used to access optional value.

```
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
       return rooms.count
    }
    subscript(i: Int) -> Room {
        get { return rooms[i] }
        set { rooms[i] = newValue }
    }
    func printNumberOfRooms() {
        print("Room count is
        \(numberOfRooms)")
    }
    var address: Address?
}

class Person {
    var residence: Residence?
}
```

```
class Address {
    var street: String?
    func streetIdentifier() -> String? {
        return street ?? nil
    }
 }
-----------------------------------------------
let john = Person()
let roomCount =
john.residence!.numberOfRooms
  // this triggers a runtime error

if let roomCount =
john.residence?.numberOfRooms {
    print("John's residence has
    \(roomCount) room(s).")
} else {
    print("Unable to retrieve the
            number of rooms.")
}
// Prints "Unable to retrieve the number
of rooms."
```

Optional Chaining

- A property's value can be set through optional chaining.
- If the optional property is nil, then attempt to set the value of any property through optional chaining will fail.
- The assignment is part of the optional chaining, which means none of the code on the right-hand side of the = operator is evaluated

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

- Optional chaining can also be used to call a method on optional values.
- Since this method is called on optional value, its return type will be void?

```
if john.residence?.printNumberOfRooms() != nil {
    print("The number of rooms printed .")
} else {
    print("The number of rooms was not printed")
}
```

- Optional chaining can be used to try to retrieve and set a value from a subscript on an optional value, and to check whether that subscript call is successful.

```
if let firstRoomName = john.residence?[0].name {
print("The first room name is \(firstRoomName).")
} else {
print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

Optional Chaining

- If a subscript returns a value of optional type—such as the key subscript of Swift's **Dictionary** type—place a question mark *after* the subscript's closing bracket to chain on its optional return value:

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80, 94, 81]
```

- Multiple level of multiple chaining can be linked together.
- However, multiple levels of optional chaining do not add more levels of optionality to the returned value.

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

- optional chaining can be used to call a method that returns a value of optional type, and to chain on that method's return value if needed.

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
   print("John's building identifier is \(buildingIdentifier).")
}
1.// Prints "John's building identifier is The Larches."
```

Error Handling

- In Swift, errors are represented by values of types that conform to the **Error** protocol.

```
enum PurchaseError: Error{
    case EmptyPurchase
    case LesserPurchase(count: Int)
}
```

- ***throws*** keyword in the function's declaration after its parameters is used to indicate that a function, method, or initializer can throw an error.

- A function which throws error is called with ***try*** keyword.

- ***do-catch*** statement is used to handle errors.
- If an error is thrown by the code in the do clause, it is matched against the catch clauses to determine which one of them can handle the error.

```
class OrderProduct{
    var quantity: Int?
    init(quantity: Int?) throws{
        guard let _quantity = quantity else {
            throw PurchaseError.EmptyPurchase
        }
        self.quantity = _quantity
    }
    func Reorder(newQuantity: Int) throws{
        guard newQuantity > self.quantity! else{
            throw PurchaseError.LesserPurchase(count:
newQuantity)
        }
        self.quantity = newQuantity
    }
}
```

Error Handling

```
do{
    let orderProduct = try
OrderProduct(quantity: nil)
    print(orderProduct.quantity!)
}catch PurchaseError.EmptyPurchase{
    print("Please Enter quantity of
product") }
```

```
do{
    let orderProduct = try OrderProduct(quantity: 10)
    try orderProduct.Reorder(newQuantity: 5)
}catch PurchaseError.EmptyPurchase{
    print("Please enetr quantity of product")
}catch PurchaseError.LesserPurchase(let newQuantity){
    print("New Order \(newQuantity) is less than previous ")}
```

- Use ***try?*** - to handle an error by converting it to an optional value
- If an error is thrown while evaluating the try? expression, the value of the expression is nil.
- Using try? lets write concise error handling code when it's desired to handle all errors in the same way

```
let productOrdered = try? OrderProduct(quantity: nil)

func fetchData() -> Data? {
  if let data = try? fetchDataFromDisk() { return data }
  if let data = try? fetchDataFromServer(){ return data }
  return nil
}
```

- ***try!*** - Is used when it's certain that function will not throw error at runtime

```
1.let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Error Handling

- A ***defer*** statement is used to execute a set of statements just before code execution leaves the current block of code.

- This statement lets to any necessary cleanup that should be performed regardless of *how* execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as **return** or **break.**

- The example uses a **defer** statement to ensure that the **open(_:)** function has a corresponding call to **close(_:)**.

```
func processFile(filename: String) throws {
    if exists(filename) {
    let file = open(filename)
    defer {
        close(file)
    }
    while let line = try file.readline() {
    // Work with the file.
    }
    // close(file) is called here, at the end of the scope.
    }
}
```

Type Casting

- *Type casting* is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.
- Type casting in Swift is implemented with the **is** and **as** operators.

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}
```

```
class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

- Create a class ***MediaItem***.
- Create other two classes ***Movie*** and ***Song*** subclassing MediaItem.
- Populate an array ***library*** with items of type Movie and Song

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
1.// the type of "library" is inferred to be [MediaItem]
```

Type Casting

- *type check operator* (**is**) is used to check whether an instance is of a certain subclass type.
- The type check operator returns **true** if the instance is of that subclass type and **false** otherwise
- ***item is Movie*** *returns* ***true*** *if the current MediaItem is a* ***Movie*** *instance and* ***false*** *if it is not. Similarly,* ***item is Song*** *checks whether the item is a* ***Song*** *instance*

```
var movieCount = 0 , songCount = 0
for item in library {
    if item is Movie { movieCount += 1}
    else if item is Song {songCount += 1} }
print("Media library contains \(movieCount)
movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3
songs"
```

- A constant or variable of certain class type can be *downcasted* to the subclass type with a *type cast operator* (as? or as!).
- The conditional form, as?, returns an optional value of the type trying to downcast to.
- The forced form, as!, attempts the downcast and force-unwraps the result as a single compound action.

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir.
\(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by
\(song.artist)")
    }
}
```